Alma Mater Studiorum · University of Bologna

Context-Aware Retrieval-based Deep Commit Message Generation

Lorenzo Borelli (lorenzo.borelli@studio.unibo.it)
Simone Mattioli(simone.mattioli6@studio.unibo.it)
Giovanni Minelli (giovanni.minelli2@studio.unibo.it

April 5, 2022

Abstract

Commit messages in version control systems aid developers in communicating changes in the source code to other developers. Oftentimes these messages are however badly formulated, or simply fail to convey the intent of the commit itself. Thus, NMT-related approaches aimed at generating commit messages from *git diffs* have begun to arise.

Contents

1 Introduction

We implemented and expanded the original method described in [1], which uses an encoder-decoder model to translate *git diffs* into commit messages. Such work was called by the authors "CoRec" that stand for "Contextual Retrieval", indeed they propose to overcome the problem of **exposure bias** and **low frequency** words, by employing respectively **decay sampling** and **retrieval-based inference**. We introduced transformers to perform the same task as CoRec, and implemented a **scheduled sampling** mechanism that adapts decay sampling for transformers.

2 Background

2.1 Automatic commit message generation

Git is a free and open source VCS designed to help develop and maintain applications, as well as allow developers to collaborate effectively. Whenever a code change is submitted through git, a commit message is included to explain the content of the change and/or the reasoning behind it. Such change is represented by a *git diff*, which contains the differences between the software versions before and after the commit. These commits are however not always meaningful or useful, and several techniques have been studied in order to automatize the commit messages generation:

- **Rule-based**: predefined rules are used to generate descriptions, they are however unable to generalize well.
- **Retrieval-based**: information retrieval approaches that use available diffs by leveraging some measure of similarity. They are limited by the absence of similar diffs in the corpus.
- **Learning-based**: techniques that learn semantic correlation between modifications in the source code and the related commit message, such as NMT models. Though efficient, they are biased towards high frequency words, being unable to truly understand words that appear rarely in the training set, and are also affected by exposure bias.

2.2 NMT

Neural Machine translation is a task that involves translating a source language into a target language. More formally, given an input sequence $x_1, ..., x_n$, and an output sequence $y_1, ..., y_n$, the aim is to maximize the probability that the target sequence is the best translation for the source sequence, that is: find the parameters of the conditional probability of the target sequence given the source $P(y|x, \theta)$. An encoder-decoder model is inherently suited to solve NMT tasks.
The encoder-decoder model involves two actors:

- **Encoder**: reads the input sequence, produces a vector representation which is given as input to the decoder.
- **Decoder**: reads the source representation in order to output the target sequence.

A RNN was the first network employed as encoder and decoder, however models like LSTM and GRU were then introduced to solve the long-term dependency problem. RNN are in fact susceptible to vanishing gradient and thus the impact of past data might be very low, whereas LSTM uses a memory cell that allows to remember important past information.

Attention is also usually applied: it is a mechanism that improves an encoder-decoder model by observing that the fixed size representation of the encoder output does not reliably capture the whole meaning of the sentence, since the decoder, for an accurate translation, might want to focus on different parts of the input representation at each step, depending on the current target token. Thus, at each decoding step, the encoder

states, together with the hidden state, are scored according to a scoring function, and then weighted to produce an attention output that is then fed back to the decoder, so that it is informed of the relative importance of each of the source tokens.

Exposure bias is a phenomenon that affects the encoder-decoder models. During training, the decoder is given the target sequence, whereas at inference time the model doesn't have the gold target sequence but, instead, the previous prediction. This means that the model is not trained under the same conditions that hold during testing, and thus it is not robust to its own errors, in fact wrong predictions might just gradually worsen the performance of the model, since they condition the decoder output. To mitigate exposure bias, a decay sampling mechanism is used, where at training time, the context for the decoder is chosen between the previous decoder's output and the ground truth, with a probability that initially favors the ground truth, and then gradually shifts to favor the previous output of the decoder that should presumably improve in its understanding of alignment.

2.3 CoRec

CoRec[1] is a seq2seq model with a few variations in both the training and testing phase. They propose a model that mixes a retrieval-based approach, in order to mitigate the problem of low frequency words, since the retrieval of the most similar diff might simply include rare tokens, and an encoder-decoder model that is able to translate changes in source code to commit messages.

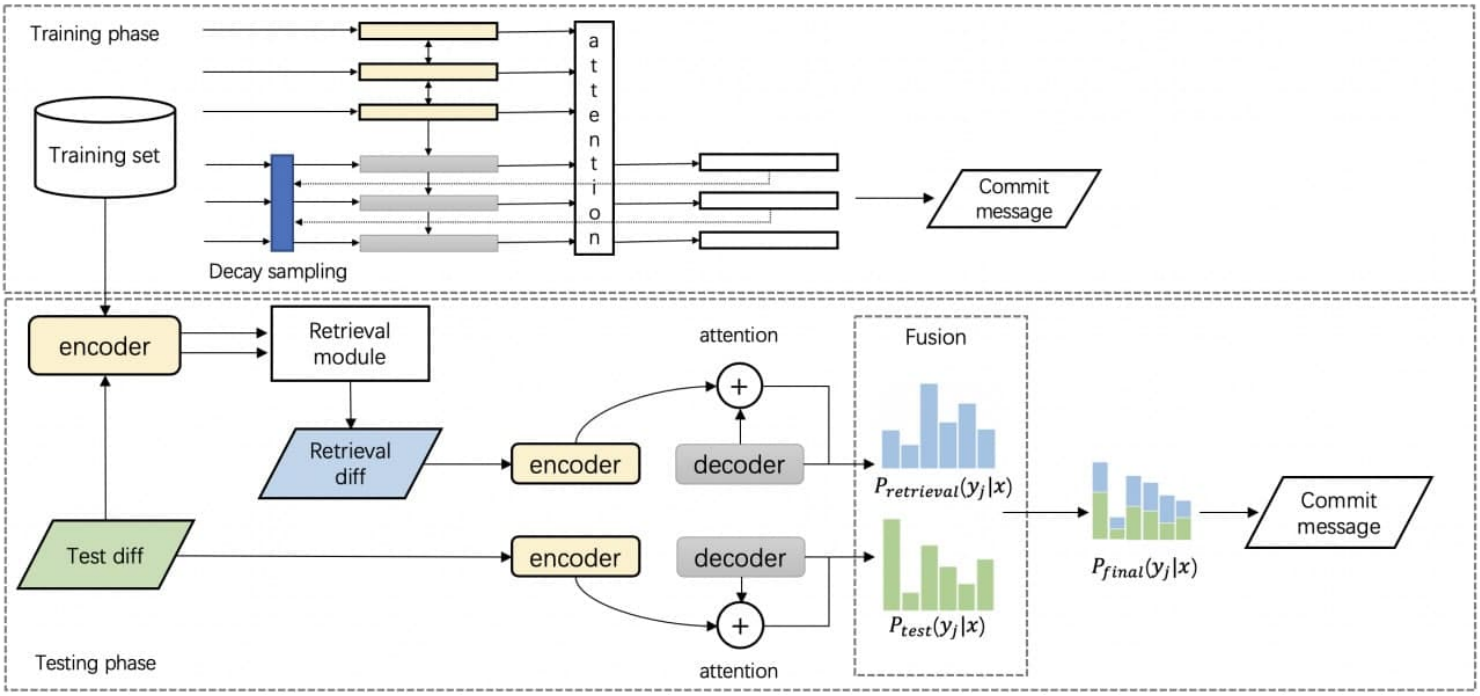

Figure 1: CoRec architecture

- **Training**: the encoder uses a Bidirectional LSTM that takes the embeddings of diffs, fixed size vector representations of each word, and applies attention to selectively focus on relevant part of the source context. The decoder is also a LSTM, whose input is decided by the decay sampling technique. The hidden state of the decoder is then used to compute the probability distribution of the next word, on all possible words in the vocabulary. The model is thus trained on the cross-entropy loss.
- **Inference**: this phase computes the final message using two probability distributions, for each word: one derives from simply computing the prediction of the test diff through the trained model, and the other is obtained by the **retrieval module**, which is the mechanism inspired by retrieval-based approaches. In order to consider low frequency words, the training diffs are passed through the trained encoder, pooled

and then the cosine similarity between them and each test diff is computed to retrieve the most similar training diff for each test diff. This semantic information is used to generate a probability distribution that is then combined with the standard probability in the following way:

$$P_{final}(y_j|x) \ = P_{test}(y_j|x) \ + \lambda BLEU(x_{retrieval}, x_{test}) P_{retrieval}(y_j|x)$$

The retrieval probability is weighted by the BLEU score, since the less similar the test diff and retrieval diff are, the less this probability should matter. Finally, **beam search** is employed to generate the final commit message: at each time step, the top *beam size* candidate sequences in terms of probability are kept until end of sentence or maximum length, and then the best one is picked.

Corec's codebase presents 3 main modules, **preprocess**, **train** and **translation**, each one dependent on the output of the previous one.

1. **Preprocess**: produces the training and validation datasets and the vocabulary with the training tokens.
2. **Train**: trains a model using the sentences in the dataset.
3. **Translate**: implements the model's inference and retrieval module.

3 Our approach

We maintained the same 3 entry-point structure as CoRec, with the following changes:

1. **Preprocess**: both source and target vocabulary and dataset creation process has been modified for compatibility with the current versions of PyTorch and Torchtext (see [5] and [6]). They are then saved for later use.
2. **Train**: the dataloader that provides the interface for iterating over batches has been rewritten. It shuffles the batches, if requested, grouping in the same batch samples that are close in length, so as to avoid too much padding. Then, it converts each batch to sequences of indices and pads them. The model is then created and trained as usual.
3. **Translate**: implements the model's inference and retrieval module. We use a modified dataset class that batches together the most similar training diffs obtained by the retrieval module, for each test sample.

3.1 Transformers

Transformers were first introduced in [2] and are the state-of-the-art in sequence-to-sequence tasks. Instead of using RNN or any variation thereof, they make use of a mechanism of **multi-head attention**, which is a combination of different "heads" of self-attention.
Self-attention, through query-key-value representations, allows at each step the encoder (and partially, the decoder) to exchange information between each token, instead of waiting for the processing of the whole sequence, which makes the transformer's training step inherently faster. More formally, the attention output is computed as

$$Attention(query, key, value) = softmax(\frac{query * key^T}{\sqrt{d_k}}) * v$$

Usually, in the decoder self-attention is masked, meaning that target tokens not yet generated are masked out, forbidding the decoder from looking ahead.
Since the transformer allows to exchange information across all tokens in a sequence at the same time step, regardless of position, there isn't an intrinsic knowledge of ordering, so transformers combine token embeddings with **position embeddings**, which are summed together with the original embeddings to obtain the actual input of the model.

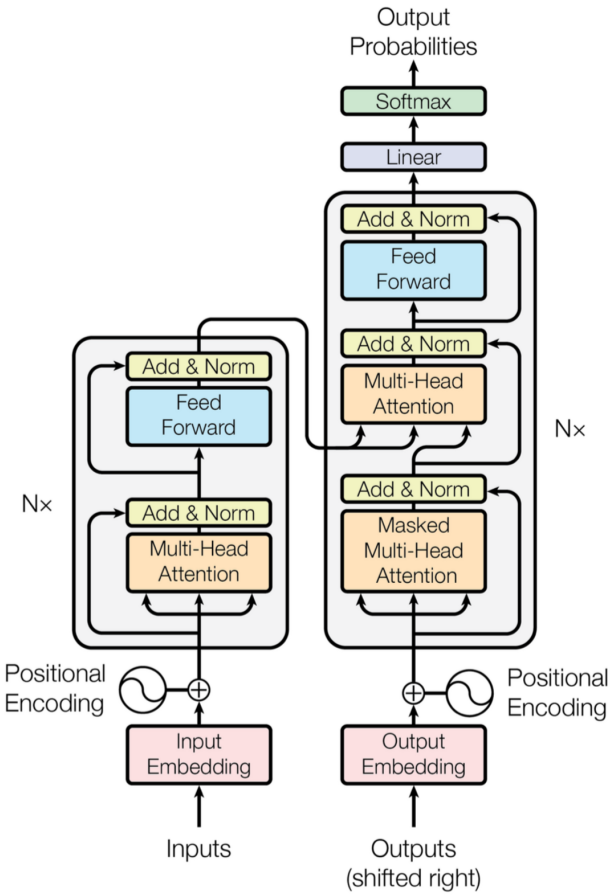

Figure 1: The Transformer - model architecture.

Figure 2: Transformer in "Attention is all you need"

Transformer architecture also features:

- **Feed-forward layer** with ReLU after each attention, to further process the tokens
- **Residual connections** between each layer and its output to ease convergence
- **Layer normalization** for stability

3.2 Scheduled sampling

Scheduled sampling is a mechanism described in [3], that adapts decay sampling to the Transformer: unlike RNN or LSTM, at each time step, Transformers generate the current word conditioned on all previous words in the sequence, making it more difficult to decide whether to use the target word or the previous output of the decoder. So, a modified architecture of the Transformer is used, with two equivalent decoders, one receiving the target representation and the other the processed output of the first or the target.

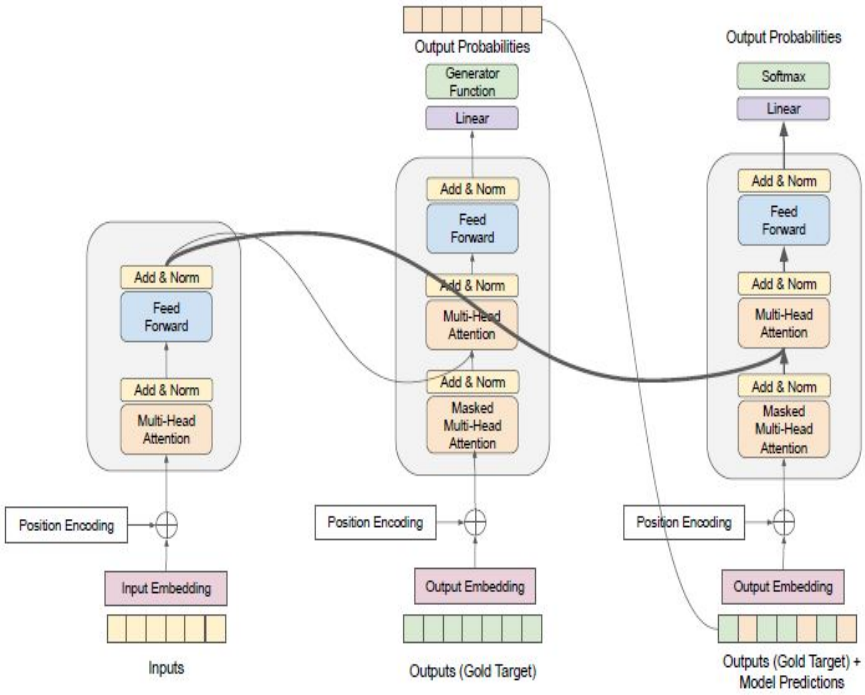

Figure 3: Scheduled sampling architecture

The workflow of the decoders is the following:

1. The first decoder simply computes the scores for each token in the vocabulary, as a standard decoder, using the golden target sequence.
2. The first decoder's predictions are processed using one of the following approaches, in order to be fed to the second decoder:
   - The *argmax* is taken for each position.
   - The weighted average of the *top-k* embeddings is selected for each position.
   - The embedding vector is the result of a *sparsemax* of the embeddings
3. The second decoder approximates scheduled sampling by taking in input a mix of the gold target sequence and the output vector of the first decoder. The probability of selecting, for each position, one of the two alternatives, is a linear function based on the training step.

4 Evaluation metrics

As metrics, we used BERTSCORE ([7]), BLEU ([8]) and METEOR ([9]):

- **Bleu**: computes n-gram overlap between predictions and reference sentences. Does not capture semantic information and can thus be deceiving. E.g. "*people like foreign cars*" and "*people like visiting places abroad*" might be scored higher than their actual semantic similarity.
- **Rouge-L**: computes the Longest Common Subsequence between reference and candidate sentences.
- **Meteor**: considers unigram alignments, but using also stemming and synonyms.
- **BERTScore**: computes similarity between predictions and reference sentences using contextualized word embeddings, which allows to take into account semantic information and distant dependencies.

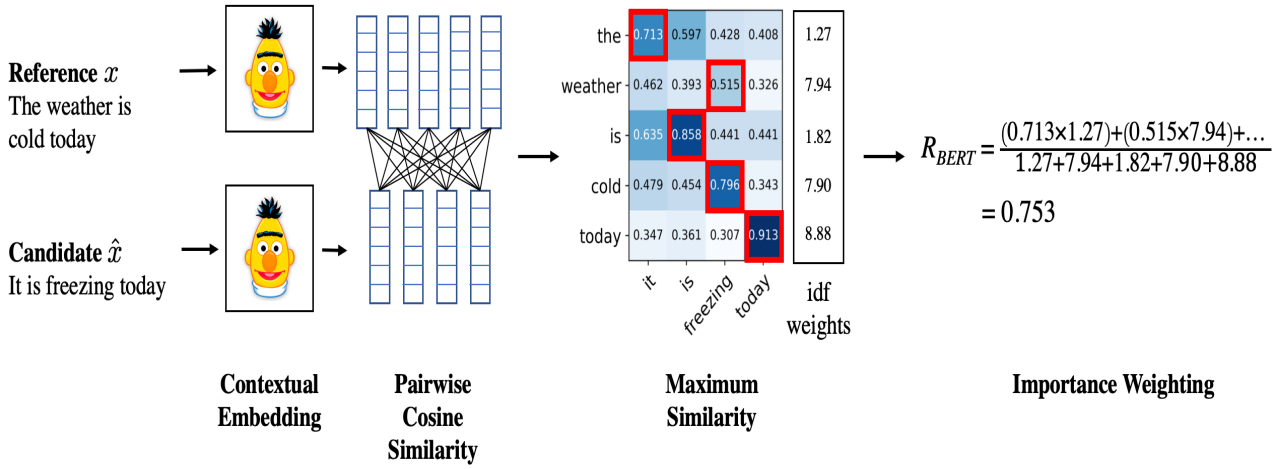

Figure 4: BERTScore

Specifically, BERTScore includes

- **Recall**: for each reference token, finds the most similar candidate token, and then divides by the total number of reference tokens
- **Precision**: for each candidate token, finds the most similar reference token, and then divides by the total number of candidate tokens
- **F1**: computed as $2\frac{P_{BERT}*R_{BERT}}{P_{BERT}+R_{BERT}}$

5 Results

We trained and tested our models on two datasets: *top1000* and *top10000*, containing respectively data from the top 1000 repos and top 10000 repos on GitHub, cleaned and parsed to respectively 22k and 98k valid commit samples. These are the same datasets used in [1].

5.1 Top1000

The amount of training data on the top1000 dataset resulted insufficient for the transformer to correctly map the validation set, indeed we have clearly spotted signs of overfitting after few dozens of steps.

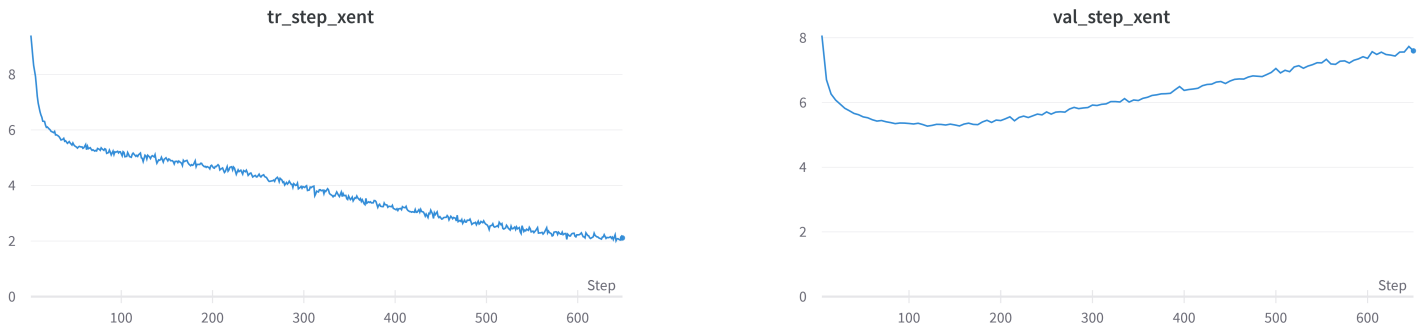

Figure 5: (a) Training and validation loss on top1000 dataset

The messages generated with an overfitted model are worse in terms of grammar and meaning utility compared to a model early stopped. We also observed that some metrics, such as Bleu and Meteor increased with the model tendency to overfit. This happened because of the trend of copying the correct words and placing them in the generated message without an effective meaning. This can easily be seen in translations like

the following, where some predictions result incredibly accurate just because they're copied, whereas others, attempt a partial copy that result in a meaningless message.

```
Prediction: updated version
Gold: updated version

Prediction: fix a test data
Gold: fixed broken test case
```

5.2 Top10000

This dataset allowed us to better observe the behavior of the transformer with different parameters. In general even with low performing model configurations we were able to reach very good results in comparison to the BiLSTM presented in [1] (results differs from the ones reported in the original paper due to differences in metric implementation). Also, Tab.1 shows that models which use retrieval are more likely to have a higher score than those that do not use it.

	Bleu	**Rouge-L**	**Meteor**	**Bert F1**
BiLSTM+ss+retrieval	**54.70%**	42.86%	39.88%	39.96%
Transformer	52.16%	43.51%	40%	40.06%
Transformer+retrieval	53.29%	**43.84%**	**40.19%**	**40.75%**
Transformer+ss	48.64%	42.25%	37.47%	37.87%
Transformer+ss+retrieval	50.79%	43.01%	38.84%	39.09%

Table 1: Table showing metrics calculated using a BiLSTM model from Corec, and transformers with combinations of scheduled sampling and retrieval.

Regarding our training process we have made some changes with respect to the original approach:

- Dropout causes an overall decrease in the training and evaluation metrics therefore we have set this value to 0.
- Learning rate decay was found to affect negatively the training process changing the slope of the validation perplexity, therefore being a meaningful parameter which could possibly have a big impact on the translations metrics we considered both situations with and without it.
- The technique adopted to reduce the exposure bias has been implemented using the Scheduled Sampling as explained in 3.2. In particular we introduced three different parameters to modify its behaviour: the number of training steps that needs to be done to start the scheduled sampling, the speed, and the minimum value for the scheduled sampling decay. We have collected data only with linear decrease approach since sigmoid and exp were advised against by the authors.

To better evaluate the effectiveness of the scheduled sampling we also collected additional data about the training phase of many runs with different configuration parameters. To draw conclusions about the results we plotted graphs of the metrics at different steps of the training process and also an heatmap showing the correlation of the training results obtained with the recorded metrics over the predicted messages.

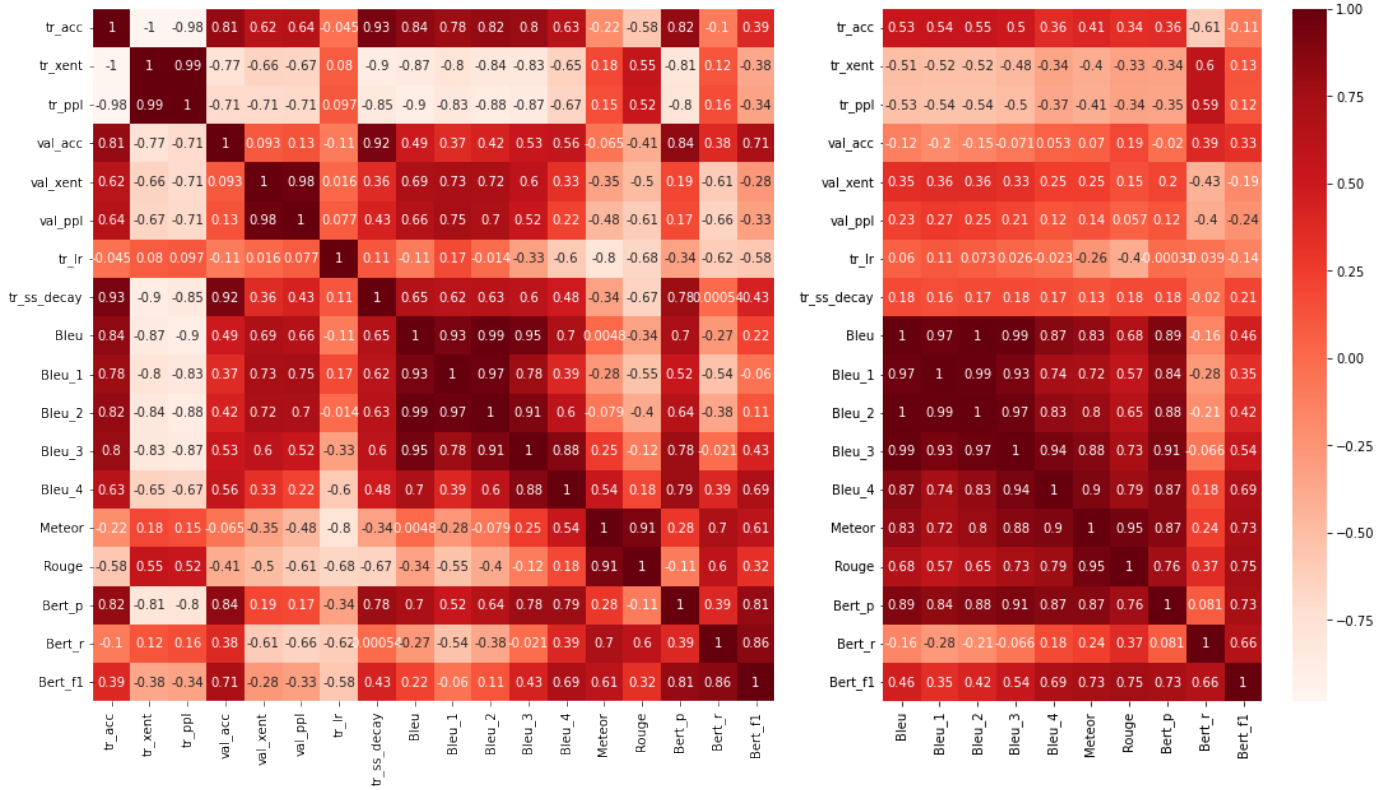

Figure 6: (a) shows the heatmap of correlation of the translation metrics with themselves and the training statistics considering the results of all combination of models, while (b) use only the results of models trained with scheduler sampling decay

Changing both scheduled sampling decay's slope, that is the speed at which we decrease the probability to switch between target and the output prediction, and the scheduled sampling minimum threshold, doesn't meaningfully influence the performance of the model, since by just increasing the number of steps the model is able to obtain equal results.

In general, we observed smaller values of validation perplexity and cross-entropy loss during training with scheduled sampling than without it, but, in addition there is a negative difference in terms of accuracy obtained. Eventually the higher accuracy both in training (+15%/20%) and in validation (+1.5%/2%) of the transformer without scheduler sampling, seems to be a much more relevant factor for the quality of the final results, w.r.t. the perplexity. Furthermore, it was noted that to keep perplexity and cross-entropy values under control was enough to decrease the training steps at a little cost of lower accuracy.
The graphs below clearly demonstrate the better scores over the models that do not use scheduled sampling.

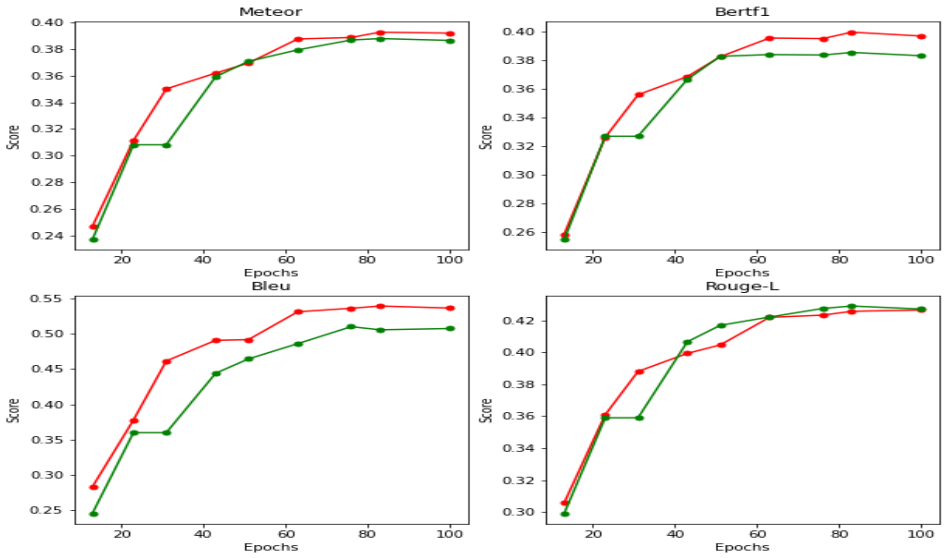

Figure 7: Plots representing Meteor, Bleu, Rouge and BERT F1 scores calculated over the results of the transformer model trained with scheduled sampling (green line) and without scheduled sampling (red line).

We can notice that after a certain step, characterized by a change of slope in validation loss and perplexity, the metrics become stable.
The Bleu results reported by models without scheduled sampling are higher due to the fact that Bleu is heavily influenced by overfitting, as shown by its positive correlation with validation perplexity and loss. Furthermore, it can be observed from Figure 6 (a), that there is an higher correlation between the learning rate and metrics like Meteor and Rouge, since decreasing the learning rate value with a decay technique reduces the slope of perplexity and therefore resulting in a better grammar structure of the sentence.
The recall value forwarded by the BertScore evaluation seems positively affected by low values in validation loss and perplexity, this was confirmed by plotting separately the correlation values for the models trained with and without scheduler sampling decay: there, we observed an evident decrease of correlation in all metrics of translation with respect to loss, accuracy, and perplexity except for that metric.
As a final thought, watching 6(b), we can say that not only the random process introduced in the training phase causes a low correlation with the metrics of translation.

5.3 Outputs

The model is generally able to correctly predict small sentences and becomes increasingly more error-prone the longer the target sentence is.
In general, we observed that low-frequency words are mostly well translated in the commit messages, e.g.

```
Prediction: modified ognlvariableexpressionevaluator constructor to protected .
Gold: modified ognlvariableexpressionevaluator constructor to protected .
```

Whenever low frequency words are present, the model also tends to memorize the target sentence word by word, which sometimes happens at the expense of contextual meaning. For example:

```
Prediction: bump spongecommon for ender dragon fix .
Gold: bump spongecommon for reusablelens fix
```

Generally, we saw polarizing results, either good or bad, which, like Jiang et al. hypothesized in [11], might be caused by the fact that the NMT model was able to predict with success messages that had a similar rationale to the ones already seen, whereas those that provided new insights were difficult to explain.

6 Conclusion

In this work we approached the task of automatic commit message generation with transformer models evaluating also scheduled sampling decay techniques. The encoder-decoder approach has also been mixed with retrieval features as suggested by similar literature that demonstrated their good influence. The implementation resulted in slight improvements in the metrics considered w.r.t. the other model proposal taken as baseline. What caught our attention during our analysis was in particular the ineffectiveness of the scheduled sampling which, in every test, worsened the final results, probably due to the introduced stochasticity in the training process. The predictions can be improved in terms of readability and more work on that task is needed before production adoption. We think that a more structured input would aid the harder translations improving the usefulness of overall outputs. Inspiring suggestions regarding such approaches can be found in Appendix B.

Appendices

A - Attention analysis

In this extension of the project, we wanted to better evaluate the attention capabilities of a transformer model applied to this task. To do so we tried different combinations of parameters at a model architecture level, different attention mechanism (functions used to compute attention), and also we applied a modification in the dictionaries of source and target words following the hint of the PosUnk approach described in [12].

A.1 Attention in the model

To better evaluate the attention mechanism of the transformers we acted on the architecture of the model by changing the number of attention heads and layers. As we already discovered in the first part of the project, increasing the number of layers is a strategy that did not affect the final result in our particular case. However, we thought that a careful analysis of what is going on under the hood would prove useful, especially making use of the attention output of the model. A debug inspection tool has been used to plot, for each layer and head, the links between the input and output sentences weighted by the value of distributed attention. From the analysis it is clearly visible how, in a multi-layered model, the same values of attention distribution are learned in the different layers thus not leading to any improvement. Moreover, we can see that many attention heads have the same pattern of attention: by decreasing the default value from 8 to 4 and eventually to 2, there is no significant loss neither in the training statistics nor in the final translation metrics, although we reduce both the number of mathematical operations and training time. Another experiment that was conducted concerns the attention mechanism: by comparing the analysis results of the attention distribution between models trained with scaled dot-product and average it is possible to see how the attention distribution focuses on multiple tokens at different steps of the generation instead of giving the most of the importance to a single element. However, this difference is not reflected in the translation metrics, whose values can be considered equivalent between the two attention mechanisms.

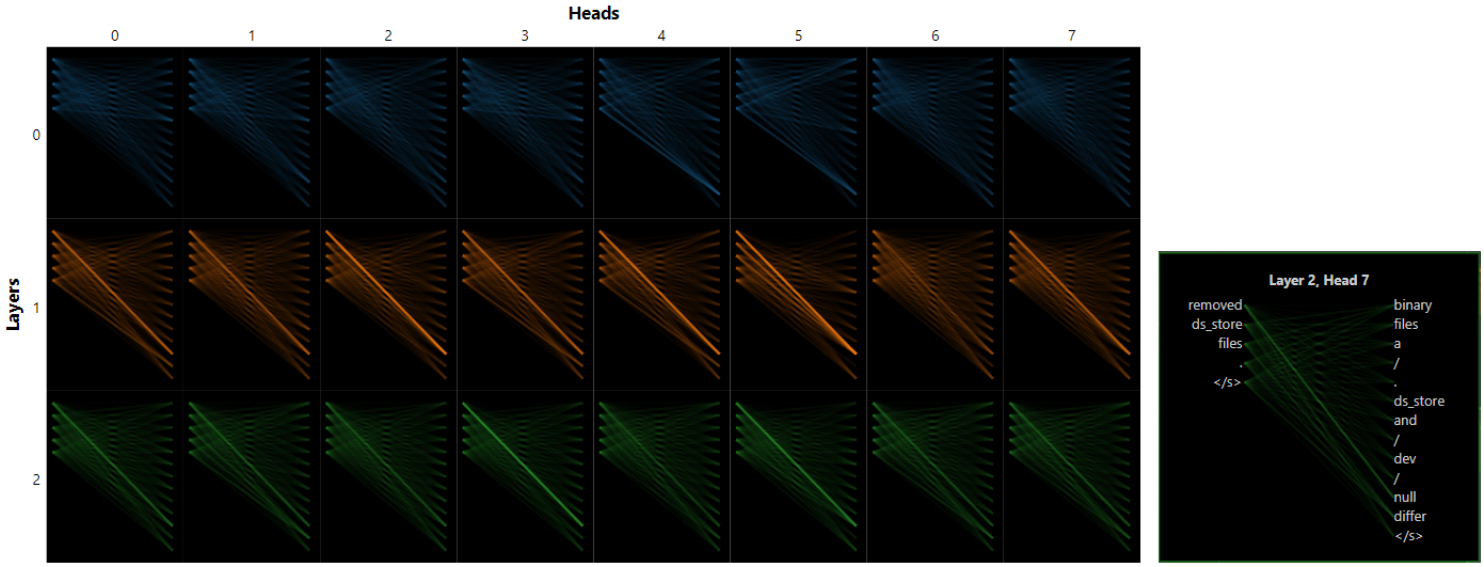

Figure 8: Visualization of the cross attention used for generation of the prediction scattered across 3 layers of 8 heads

A.2 Attention at translation time

Since the model already generates an attention distribution during the prediction of the sentence, ideally we can make use of it at prediction time to favor less frequent words during decoding at virtually zero cost.
A first test using this idea consisted of a very simple decrease in the target vocabulary: a decrease in the

output embedding layer also results in a large reduction in the number of parameters. The vocabulary was filtered with a dictionary of common English words, going from 36k tokens to only 13k. This was done in order to somehow let the model only learn associations to create good sentences but not be able to generate words such as function names or libraries that can be found in the training set. The input would be encoded with '¡unk¿' tokens in spite of those words, and as result the predicted sentences would obviously contain a lot of unknowns. In order to recover the words to replace the unknowns we rely on the attention mechanism which give us a distribution of values for each input token in the current generation step. The tokens with the higher values are extracted and the first one corresponding to a useful token (e.g. punctuation is not considered useful) is used as replacement in the generated sentence.
The evaluation has been performed on both the RNN model of CoRec and on our Transformer implementation obtaining equivalent findings, therefore we considered as baseline our best model of Transformers with the Retrieval module. The obvious result is an increase in training accuracy since fewer associations have to be learned by our model; in terms of translation metrics the results are reasonable (Table A.2), but from a human review we can observe many sentences losing their meaning. In particular, we can see that when the model produces an ineffective sentence prediction, even the attention result is not correctly distributed:

```
Gold: remove an extra new line
Baseline pred: add a private constructor to preconditionmessage
Reduced pred: add a private constructor to cannot
```

When the sentences generated are short and only 1 word is picked with attention the approach result useful, while, when we are dealing with longer sentences and multiple unknowns are present, could happen that the attention focuses on a single token:

```
Gold: add mandatory ‘ rule_name ‘ attribute to jvm_maven_import_external . . .
Baseline pred: add docstring tests to buck_module .
Reduced pred: make java_import use java_import instead of java_import . . . .
```

The second test was about an even decrease in both source and target vocabularies, in a manner more similar to that presented by [12]. Instead of letting the model learn many associations between a commit diff and a message where just names of functions and class might change, we can ease the learning process to the model by replacing all these words with a set of tokens like pos0 ..pos99 then recovering the correspondence between source and target at translation time using the attention as previously described.
The tokens removed were 14346 out of the 25k in common between source and target, being the only words not present in the English dictionary considered. The results A.2 this time obtained a smaller decrease in bleu, in particular bleu 1gram appears little affected compared to the baseline value. The readability of the predicted sentences is quite good, but still suffers from the problem highlighted earlier, where the same input word gets most of the attention at different steps of generation, and thus resulting in generated messages that present the same word many times.

(Transformer + retrieval)	**Parameters**	**Bleu**	**Rouge-L**	**Meteor**	**Bert F1**
Baseline	126235220	53.29%	43.84%	40.19%	40.75%
Targets reduced	102137983(-19.1%)	48.69%	41.06%	36.60%	34.19%
Pos_unk reduction	104339118 (-17.3%)	50.8%	41.73%	37.67%	36.67%

Table 2: Results of the evaluation metrics obtained from models trained with reduced vocabularies

B - Survey on methods for automatic code summarization

In this review, we focused on approaches related to the world of deep learning models, given recent years' advancements and results, while still providing an overview of methods used to solve this problem, through the most relevant papers.

B.1 Overview

According to [1], techniques for code summarization can be generally categorized in

- **Rule-based**: predefined rules/templates are used to generate descriptions.
- **Retrieval-based**: information retrieval approaches try different techniques to extrapolate relevant data from source code and generate a description from that data.
- **Learning-based**: techniques that learn semantic correlation between source code and description.

B.2 Rule-based

Predefined templates are created by developers and researchers, based on features related to structure of the program. Several approaches are used to create these templates.
[17] produces descriptions of the form *action theme (secondary arguments)* for method calls like *receiver.verbNoun(arg)* where the action is usually the verb. The paper applies this approach to *object-related action units*, a code block that performs a high level algorithmic step, whose instructions are associated with each other by objects, e.g.

```
TableRow mappingRow = DatabaseManager.create(
    ourContext, "bundle2bitstream");
mappingRow.setColumn("bundle_id", getID());
mappingRow.setColumn("bitstream_id", b.getID());
database.add(mappingRow);
```

Abid et al. in [16] use stereotypes, that is abstractions of classes' and functions' roles (like factory, predicate, set..), generated by static analysis, and a set of predefined templates to write descriptions of C++ code.

Rule-based methods suffer from poor generalization, due to the set of assumptions they need in order to work, like naming conventions, code quality, language, pragmatics.

B.3 Retrieval-based

These methods use Information Retrieval techniques to find salient information in source code and translate it to a natural language description.
[15] defines two types of information extraction: extractive and abstractive. The former are term-based summaries that simply highlight keywords in the code, whereas the latter gives a more detailed explanation. The paper then develops a framework to select the most relevant terms in a document, where a document could be a file, a class, a package. Different algorithms are tested to find these terms, like Latent Semantic Indexing and Vector Space Model, with different combinations of weighting techniques.
Papers like [14] implemented a retrieval-based technique that consists in finding description-code pairs and use them to select descriptions for new code fragments, based on some measure of similarity.

Retrieval-based methods are too dependent on the presence of similar code snippets, as well as the employment of good code practices, which are sometimes mandatory in order for the retrieval algorithms to work.

B.4 Learning-based

This family of methods entails training a model to learn the underlying relation between source code and natural language description.
First attempts were made with supervised and unsupervised learning, which both require feature engineering; the growth of artificial neural network quickly highlighted the possibility to avoid the process of manually selecting features, which, like [23] explains, can be especially difficult in a setting with lots of heterogeneous data.

B.5 NMT for code summarization

The first to propose an NMT-based approach to code changes were Loyola et al. in [23], following the intuition that there might be a way to develop a model able to explain source level modifications in natural language. With this premise, the NMT task, in this context, can be seen as a translation from the language of the source code to the natural language description. Loyola et al. implemented an LSTM encoder-decoder architecture, with attention, and beam search to output the best translation.

Further studies tackled some of the shortcomings of such an approach: Liu et Al. in [24] developed *PtrGNCMsg*, a modified encoder-decoder model that takes into account the proliferation of OOV tokens in code summarization, given that code inherently features many context-specific identifiers. It uses a *PointerGenerator* network, that decides at each decoder step whether to output the next word from the vocabulary or the next word in the input sequence (through a probability conditioned by attention). Siyuan et al. in [11] observed that the quality of commit messages could be improved by filtering only those sample messages that followed a *verb-direct object* grammatical pattern. This filter was also applied by CoRec ([1]).
In [20], the authors pointed out that, while comment generation (and in general code summarization) can be seen as a NMT task, code structure is inherently different from natural languages, thus requiring dedicated techniques to process it, in order to retain its structural information, which is fundamental to truly understand code: they used an AST representation of code, and invented a new type of visit (Structure based traversal) of the tree that would allow to maintain the information about structure during conversion for the model; moreover, they replaced the universal *UNK* for unknown tokens with their type, in order to mitigate the problem of OOV proliferation.
[13] proposes a different method of encoding the linearized AST input of the transformer, namely Relation Matrices, which only include information, for each node of the tree, about the direct ancestor, descendant and siblings (instead of every pair of nodes), in order to limit the size of the input, reducing overhead and long-range dependency problems.
[19] and [18] introduce pre-trained contextual embeddings: the former uses a two-phase transformer, during first phase to create contextual word representations, training on code-before and code-after-change pairs, and the second phase is used to fine-tune the transformer on the code-message pairs with the found embeddings; the latter uses CodeBERT and differentiates between code additions and deletions.

Paper	Key concepts	Methods, Data	Metrics, Results
A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes	Non-NMT methods are too static. Need for a model able to understand source code level modifications. First attempt to avoid feature engineering.	Encoder-decoder LSTM with attention. Beam search. A dataset of Python, Java, C++, Javascript diffs.	Bleu-4, Meteor. Affected by memorization.
Automatically generating Commits Messages from Diffs using Neural Machine Translation	Automatically generated commits explain the how but not why. Good messages can be learned with sufficient and filtered data.	Encoder-decoder LSTM with attention. V-DO filters to messages, and only first sentence of commit taken as target. Only Java files.	Bleu and human evaluation, which provide label for SVM classifier.
Generating commit messages from Diffs using Pointer-Generator networks	Code contains lots of context-specific identifiers, so need mechanism to handle OOV	Pointer-generator network to compute tradeoff between next word from vocabulary and from input sequence.	Bleu and Rouge. Risks overfitting.
Deep code comment generation	Code has different structure from natural language and thus different techniques must be used to process it.	Encoder-decoder model fed with AST-linearized code. SBT technique for traversal. Type annotations for oov. Java code	Bleu-4, better comprehension of semantics thanks to AST, but still tested on Java only.
AST-Transformer: Encoding Abstract Syntax Trees Efficiently for Code Summarization	AST encoding is very long, causing performance issues.	Encoder-decoder architecture with Transformer, Relation Matrices for AST encoding, Multi-Head attention for each relation matrix. Tested on Java and Python datasets	Bleu, Rouge and Meteor. Outperforms baseline with standard AST linearization.
CoreGen: Contextualized code representation Learning for Commit Message generation	Code tokens change meaning depending on context. Also, there are different type of commits.	Train a transformer to learn contextualized embeddings, handle source commits and binary file commits differently.	Bleu-4, rouge and meteor
CommitBERT: Commit Message Generation using Pre-Trained Programming Language Model	Addition and deletion of code must be differentiated. CodeBERT is a pre-trained programming language model especially suited for this task.	Encoder-decoder model with CodeBERT to initialize weights, and as input pairs of *addition,deletion* from repositories of 6 different languages.	Bleu-4, better results than with random initialization and RoBERTa

Table 3: NMT methods

References

[1] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. *Context-Aware Retrieval based Deep Commit Message Generation*. ACM Trans. Softw. Eng. Methodol. 1, 1 May 2021. 10.1145/nnnnnnn.nnnnnnn

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin *Attention Is All You Need* NIPS 2017, Vol 30, https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[3] Tsvetomila Mihaylova, André F. T. Martins *Scheduled Sampling for Transformers*. arXiv:1906.07651v1 [cs.CL] 18 Jun 2019

[4] Git https://git-scm.com/

[5] PyTorch https://pytorch.org/

[6] Torchtext https://pytorch.org/text/stable/index.html

[7] Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinbergeryz, and Yoav Artzizx *BertScore: Evaluating Text Generation With Bert* arXiv:1904.09675v3 24 Feb 2020

[8] Kishore Papineni, Salim Roukos, Todd Ward, Wei-Jing Zhu *BLEU: a method for automatic evaluation of machine translation* ACL '02: Proceedings of the 40th Annual Meeting on Association for Computational LinguisticsJuly 2002 Pages 311–318, doi: 10.3115/1073083.1073135

[9] Satanjeev Banerjee, Alon Lavie *METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments* Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, Jun 2005

[10] BERTScore GitHub https://github.com/Tiiiger/bert_score

[11] Siyuan Jiang, Ameer Armaly, and Collin McMillan *Automatically generating Commits Messages from Diffs using Neural Machine Translation* 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp. 135-146, doi: 10.1109/ASE.2017.8115626

[12] Thang Luong, Ilya Sutskever, Quoc Le, Oriol Vinyals, and Wojciech Zaremba. 2015. *Addressing the Rare Word Problem in Neural Machine Translation* In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 11–19, Beijing, China. Association for Computational Linguistics. https://aclanthology.org/P15-1002

[13] Ze Tang, Chuanyi Li, Jidong Ge, Xiaoyu Shen, Zheling Zhu, Bin Luo *AST-Transformer: Encoding Abstract Syntax Trees Efficiently for Code Summarization*

[14] Y. Huang, Q. Zheng, X. Chen, Y. Xiong, Z. Liu, and X. Luo, *Mining version control system for automatically generating commit comment*, in Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. IEEE Press, 2017, pp. 414–423.

[15] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, *On the use of automated text summarization techniques for summarizing source code* in 2010 17th Working Conference on Reverse Engineering. IEEE, 2010, pp. 35–44.

[16] N. J. Abid, N. Dragan, M. L. Collard and J. I. Maletic, *Using stereotypes in the automatic generation of natural language summaries for C++ methods*, 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 561-565, doi: 10.1109/ICSM.2015.7332514.

[17] X. Wang, L. Pollock, and K. Vijay-Shanker, *Automatically generating natural language descriptions for object-related statement sequences*, in 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2017, pp. 205–216.

[18] Tae-Hwan Jung *CommitBERT: Commit Message Generation Using Pre-Trained Programming Language Model* Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021), pages 26–33 August 1–6, 2021. ©2021 Association for Computational Linguistics

[19] Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, Zenglin Xu, *CoreGen: Contextualized Code Representation Learning for Commit Message Generation*, Neurocomputing, Volume 459, 12 October 2021, Pages 97-107

[20] X. Hu, G. Li, X. Xia, D. Lo and Z. Jin, *Deep Code Comment Generation*, 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), 2018, pp. 200-20010.

[21] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, *Summarizing source code using a neural attention model*, in Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, pp. 2073–2083. (PRIMO using neural networks)

[22] Nazar, N., Jiang, H., Gao, G. et al. *Source code fragment summarization with small-scale crowdsourcing based features*. Front. Comput. Sci. 10, 504–517 (2016). https://doi.org/10.1007/s11704-015-4409-2

[23] Pablo Loyola, Edison Marrese-Taylor and Yutaka Matsuo, *A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes*, arXiv:1704.04856v1 [cs.CL], 17 Apr 2017

[24] Q. Liu, Z. Liu, H. Zhu, H. Fan, B. Du and Y. Qian, *Generating Commit Messages from Diffs using Pointer-Generator Network*, 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019, pp. 299-309, doi: 10.1109/MSR.2019.00056.